Fluent Model Checking for Event-based Systems

Dimitra Giannakopoulou
RIACS/USRA, NASA Ames Research Center,
Moffett Field, CA 94035-1000, USA
dimitra@email.arc.nasa.gov

Jeff Magee
Department of Computing, Imperial College London,
180 Queen's Gate, London SW7 2AZ, UK
jnm@doc.ic.ac.uk

ABSTRACT

Model checking is an automated technique for verifying that a system satisfies a set of required properties. Such properties are typically expressed as temporal logic formulas, in which atomic propositions are predicates over state variables of the system. In event-based system descriptions, states are not characterized by state variables, but rather by the behavior that originates in these states in terms of actions. In this context, it is natural for temporal formulas to be built from atomic propositions that are predicates on the occurrence of actions. The paper identifies limitations in this approach and introduces "fluent" propositions that permit formulas to naturally express properties that combine state and action. A *fluent* is a property of the world that holds after it is initiated by an action and ceases to hold when terminated by another action. The paper describes an approach to model checking fluent-based linear-temporal logic properties, with its implementation and application in the LTSA tool.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Model checking;

General Terms

Design, Languages, Verification.

Keywords

Model-checking, linear temporal logic, software architecture analysis.

1. INTRODUCTION

Our work is targeted at analyzing the behavioral properties of complex systems at the architectural level. In this context, the software architecture of a system is described by a set of components, the structure that interconnects these components and the connectors that describe how components interact. In common with other researchers concerned with the rigorous specification of software architectures, we have chosen an event-based formalism to describe behavior. This is a natural choice since at the architectural level, we are concerned with the interactions between components in terms of messages sent or received by components and service invocations initiated or accepted by components. These messages and invocations are most naturally modeled as actions or events, and component behavior is thus modeled in terms of these actions and events (in the rest of the paper, the terms event and action will be used interchangeably). For example, the Architectural Description Language (ADL) Wright [1] is based on the process algebra CSP[15] for behavior descriptions and FDR[31] for behavior analysis; PADL[4] is based on CCS[26] and via TwoTowers, uses the Concurrency Workbench [7] for functional analysis; our own Darwin ADL[20] uses the process algebra style language FSP [21], and the Labeled Transition System Analyzer (LTSA) tool [22].

In these event-based models of behavior, properties are described in terms of actions. Our experience however has been that, although in some cases this is a natural approach to take, the task of expressing properties often becomes unmanageable. To address this problem, we introduce "fluent" propositions, which define state predicates whose values are determined by the occurrence of actions. Fluents provide an elegant and uniform framework for supporting the description of properties that combine state and action. Moreover, we have developed an approach for model checking fluent-based linear-time temporal logic (LTL) properties.

Although fluents are applicable to other temporal logics, we are interested in LTL for a number of reasons. We find it natural to reason about component interactions in terms of sequences of events in linear time. LTL is an expressive formalism that is also supported by efficient techniques for on-the-fly model checking. In addition, LTL is one of the logics that is supported by the work on specification patterns [10], which we wish to utilize in our framework for providing users with assistance in expressing properties formally. This body of work has undoubtedly been inspired by the success and popularity of the SPIN model-checking tool[16] that incorporates many of the recent advances in LTL model checking.

Why not use existing state-based model checkers such as SPIN to analyze Software Architecture? The reasons are twofold. Firstly, we wish to retain the compositional character of process algebra descriptions in which sub-components can be composed to form behaviorally equivalent components with reduced state space. In process algebra, component behavior is specified directly in terms of actions and local state is not explicitly represented. Secondly, the atomic propositions of LTL properties in SPIN are predicates on state variables. Properties on events can only be specified indirectly in relation to changes in these state variables. A recent paper [27] presents an approach to specifying events in LTL properties. However, this approach still has the problem that events are specified indirectly in terms of edges. These edges are changes to the truth values of atomic propositions and relate to state variables rather than directly to actions.

ESEC/FSE'03, September 1–9, 2003, Helsinki, Finland

In the following, Section 2 presents some background on LTL model checking. Section 3 presents an interpretation for LTL with actions as atomic propositions, and illustrates the limitations of this approach. A more flexible approach is developed in Section 4 using *fluents*. The basic model checking procedure for LTL based on fluents (FLTL) is outlined in Section 5, and is subsequently refined in Section 6. Section 7 discusses the implementation of FLTL model checking in the LTSA tool, and Section 8 closes the paper with conclusions and future directions.

2. BACKGROUND

This section provides some background on Labeled Transitions Systems(LTSs), Linear-time Temporal Logic (LTL), and the automata-theoretic approach to model checking LTL properties.

2.1 Labeled Transition Systems (LTSs)

We use LTSs to model the behavior of communicating components in a concurrent system. Let *Act* be the universal set of observable actions, and let τ denote a local action that is *unobservable* to a component's environment. An LTS M is a quadruple $\langle Q, A, \delta, q_0 \rangle$ where:

- Q is a finite set of states,
- $A \subseteq Act$ is the communicating *alphabet* of M,
- $\delta \subseteq Q \times A \cup \{\tau\} \times Q$ is a labeled transition relation,
- $q_0 \in Q$ is the initial state.

We call an *execution* of M a sequence of actions (observable or τ) that M can perform starting at its initial state. We say that $M = \langle Q, A, \delta, q_0 \rangle$ transits into $M' = \langle Q, A, \delta, q_0' \rangle$ with action a, denoted as $M \xrightarrow{a} M'$, if and only if $(q_0, a, q_0') \in \delta$.

The *hiding* operator "\" takes an LTS $M = \langle Q, A, \delta, q_0 \rangle$ and a set of actions $H \subseteq Act$, and returns $M \setminus H = \langle Q, A - H, \delta', q_0 \rangle$, where δ' is obtained from δ by turning all transitions labeled with actions in H into τ transitions.

The *parallel composition* operator "||" is a commutative and associative operator that combines the behavior of two LTSs by synchronizing the actions common to their alphabets and interleaving the remaining actions. Let $M_1 = \langle Q^1, A^1, \delta^1, q_0^1 \rangle$ and $M_2 = \langle Q^2, A^2, \delta^2, q_0^2 \rangle$. Then $M_1 \| M_2 = \langle Q, A, \delta, q_0 \rangle$, where $Q = Q^1 \times Q^2$, $A = A^1 \cup A^2$, $q_0 = (q_0^1, q_0^2)$, and δ is defined as follows, where a is an observable action or τ (the symmetric rules are implied by commutativity of ||):

- $$\frac{M_1 \xrightarrow{a} M_1', a \notin A^2}{M_1 \| M_2 \xrightarrow{a} M_1' \| M_2}$$
- $$\frac{M_1 \xrightarrow{a} M_1', M_2 \xrightarrow{a} M_2', a \neq \tau}{M_1 \| M_2 \xrightarrow{a} M_1' \| M_2'}$$

2.2 Linear Temporal Logic (LTL)

Given a set of atomic propositions $\wp$, a well-formed LTL formula is defined inductively using the standard Boolean operators, and the temporal operators **X** (next) and **U** (strong until) as follows:

- each member of $\wp$ is a formula,
- if φ and ψ are formulas, then so are ¬ φ, φ ∨ ψ, φ ∧ ψ, **X**φ, φ **U** ψ.

An interpretation for an LTL formula is an infinite word $w = x_0 x_1 x_2 \ldots$ over $2^{\wp}$. In other words, an interpretation maps to each instant of time a set of propositions that hold at that instant. We write w_i for the suffix of w starting at x_i. LTL semantics is then defined inductively as follows:

- $w \models p$ iff $p \in x_0$, for $p \in \wp$
- $w \models \varphi \vee \psi$ iff ($w \models \varphi$) or ($w \models \psi$)
- $w \models \neg \varphi$ iff not $w \models \varphi$
- $w \models \varphi \wedge \psi$ iff ($w \models \varphi$) and ($w \models \psi$)
- $w \models \mathbf{X} \varphi$ iff $w_1 \models \varphi$
- $w \models \varphi \mathbf{U} \psi$ iff $\exists\, i \geq 0$, such that:
  $w_i \models \psi$ and $\forall\, 0 \leq j < i$, $w_j \models \varphi$

We introduce the abbreviations "true ≡ φ ∨ ¬φ" and "false ≡ ¬true". Boolean operator ⇒ is defined as follows: φ ⇒ ψ ≡ ¬φ ∨ ψ. Temporal operators **F** (eventually), **G** (always), and **W** (weak until) are defined in terms of the main temporal operators as follows: **F**φ ≡ true **U** φ, **G**φ ≡ ¬**F**¬φ, and φ **W** ψ ≡ ((φ **U** ψ) ∨ **G**φ).

2.3 Model checking LTL

Model-checking, invented by Clarke and Emerson[6] and Queille and Sifakis [29], is an automated technique for checking a finite state system against some temporal logic specification. The standard automata-theoretic approach [35] to model checking LTL properties is based on the use of Büchi automata.

A *Büchi automaton (BA)* is a 5-tuple $B = \langle Q, \Sigma, \delta, q_0, F \rangle$, where Q is a finite set of states, Σ is a finite set of labels, $\delta \subseteq Q \times \Sigma \times Q$ is a labeled transition relation, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is a set of accepting states.

An *execution* of B on an infinite word $< a_0 a_1 a_2 \cdots >$ over Σ is an infinite word $< s_0 s_1 s_2 \cdots >$ over Q, such that: $s_0 = q_0$ and $\forall\, i \in N$, $(s_i, a_i, s_{i+1}) \in \delta$. An execution is accepting if some element of F occurs in it infinitely often. An infinite word w over Σ is *accepted* by the automaton B, if there exists some accepting execution of B on w.

For any LTL formula over a set of propositions $\wp$, a Büchi automaton can be constructed that accepts exactly those infinite words over $2^{\wp}$ that satisfy φ. Based on this result, a finite-state system is checked against an LTL specification φ by computing the intersection of the system with a Büchi automaton corresponding to (¬φ); the system satisfies φ if this intersection accepts no words, i.e., if its language is empty.

3. ACTION LTL (ALTL)

In the context of event-based behavior specifications, we must determine the meaning of the atomic propositions from which LTL formulas are built. Our first approach, following from the work of one of the authors[12] and used by Leuschel, Massart and Currie[24] in relation to CSP, is to directly associate propositions with actions. We term this approach ALTL to distinguish it both

from the usual state-based interpretations for LTL and from our subsequent approach based on fluents.

In ALTL, the set of propositions $\wp$ is the universal set of actions (or events) *Act*. An interpretation assigns to each time instant a set of actions that occur at that instant. In our interleaving model of concurrency where a single action can occur at a time, these sets are singletons. An infinite execution $< a_0 a_1 a_2 \cdots >$ of an LTS defines an interpretation that assigns action a_i to each time instant $i \in N$. An LTS *M satisfies* an LTL property φ if and only if φ holds in all interpretations defined by executions of *M*. Note that, unlike standard trace theory [15], τ actions must be taken into account in this context, because they may be the cause of *infinite* stuttering, identified in process algebras as *divergence*. Initially, to simplify the presentation, we will assume that all actions in the system LTS are observable. We deal with LTSs with τ actions in Section 6.

3.1 Limitation

Although ALTL provides a clear and obvious interpretation for LTL formulas in the context of event-based system descriptions, it is limited in the ease with which properties can be specified. To illustrate this, we use an example arising from work on verifying a new decentralized style for organizing television control software[34]. Figure 1 shows a simplified fragment of the software architecture of a television organized using this style. Each component is responsible for controlling a part of the signal path in the television.

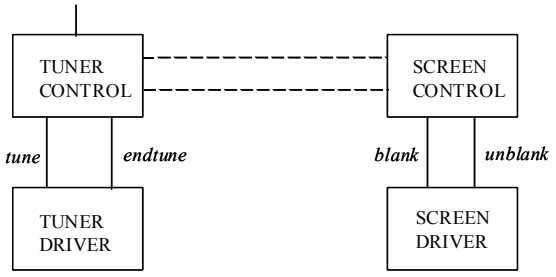

Figure 1. Fragment of TV software architecture

A required property of this system is that to avoid artifacts on the screen when changing channels, the screen must be blanked when the tuner is tuning into the new channel. The screen is blanked by the action *blank* and displays the new channel signal when the action *unblank* occurs. The tuner starts tuning into the new channel when initiated by the action *tune* and indicates that it has finished tuning by the action *endtune*. The required safety property can be stated quite simply in English: *"If the tuner is tuning, then the screen must be blanked."* However, translating this into ALTL is less straightforward. If we make the assumption that the screen is initially blanked, the required property can be expressed as:

$$\text{NOARTIFACTS} \equiv \mathbf{G}(\ (\mathit{unblank} \Rightarrow (\neg \mathit{tune}\ \mathbf{W}\ \mathit{blank})) \ \wedge \ (\mathit{tune} \Rightarrow (\neg \mathit{unblank}\ \mathbf{W}\ \mathit{endtune})))$$

If we assume that the screen is not initially blanked then the property becomes:

$$(\neg \mathit{tune}\ \mathbf{W}\ \mathit{blank}) \wedge \mathbf{G}(\ (\mathit{unblank} \Rightarrow (\neg \mathit{tune}\ \mathbf{W}\ \mathit{blank})) \ \wedge \ (\mathit{tune} \Rightarrow (\neg \mathit{unblank}\ \mathbf{W}\ \mathit{endtune})))$$

In the actual TV model, there is more than one method of initiating tuning and blanking and more than one way that these activities and states terminate. In addition, the property must be specified for architectures with multiple tuners and multiple output devices. The task of specifying the required property in ALTL quickly becomes unmanageable. The reason is that ALTL is limited to action propositions (only one of which can be true at any given instant). It is often non-trivial to express LTL properties directly in terms of actions, especially when these actions are used to define time intervals and relationships between them (e.g. no overlapping). Such intervals are often easier to define in terms of values of (system) state predicates that characterize them. For example, if we could observe a predicate *TUNING* of the tuner driver which was true when the driver was tuning and a predicate *BLANKED* of the screen driver which was true when the screen was blanked, we could express the desired property in LTL simply as $\mathbf{G}(\mathit{TUNING} \Rightarrow \mathit{BLANKED})$. This definition is a direct translation of the requirement stated in English above. The problem of observing state predicates in event-based system descriptions is addressed in the next section.

4. FLUENT LTL (FLTL)

To reason about the effects of actions on the state of a system, we adopt the idea of a *fluent* from research in Artificial Intelligence[32]. A fluent is anything whose value is subject to change over time, although here we deal only with propositional fluents. The notion is best explained by an informal definition due to Miller and Shanahan[25] from the Event Calculus originally introduced by Kowalski and Sergot[18] as a logic program framework for reasoning about actions and their effects. Miller and Shanahan informally define propositional fluents as follows: "Fluents (time-varying properties of the world) are true at particular time-points if they have been initiated by an action occurrence at some earlier time-point, and not terminated by another action occurrence in the meantime. Similarly, a fluent is false at a particular time-point if it has been previously terminated and not initiated in the meantime."

Our use of fluent is consistent with this definition. We define a fluent *Fl* by a pair of sets, a set of initiating actions I_{Fl} and a set of terminating actions T_{Fl}:

$Fl \equiv \langle\, I_{Fl}\,,\, T_{Fl}\,\rangle$ where $I_{Fl}\,,\, T_{Fl} \subset \mathit{Act}$ and $I_{Fl} \cap T_{Fl} = \varnothing$

In addition, a fluent *Fl* may initially be true or false at time zero as denoted by the attribute $\mathit{Initially}_{Fl}$.

The set of atomic propositions from which FLTL formulas are built is the set of fluents Φ. Therefore, an interpretation in FLTL is an infinite word over 2^{Φ}, which assigns to each time instant the set of fluents that hold at that time instant. Similarly to ALTL, any infinite word $< a_0 a_1 a_2 \cdots >$ over *Act* also defines an FLTL interpretation $< f_0 f_1 f_2 \cdots >$ over 2^{Φ} as follows:

$\forall i \in N, \forall\, Fl \in \Phi, Fl \in f_i$ iff either of the following holds

- $\mathit{Initially}_{Fl} \wedge (\forall k \in N \cdot 0 \le k \le i, a_k \notin T_{Fl})$
- $\exists j \in N : ((j \le i) \wedge (a_j \in I_{Fl}) \wedge (\forall k \in N \cdot j < k \le i, a_k \notin T_{Fl}))$

In other words, a fluent holds at a time instant if and only if it holds initially or some initiating action has occurred, and in both

cases, no terminating action has yet occurred. Note that the interval over which a fluent holds is *closed* on the left and *open* on the right, since actions have immediate effect on the values of fluents. This is slightly different from the fluents of Miller and Shanahan, which are open on the left and closed on the right[25]. Since the sets of initiating and terminating actions are disjoint, the value of a fluent is always deterministic with respect to a system execution. Note that the frame problem as it relates to fluents in the Event Calculus[25] is not relevant here as we have a fixed set of well defined actions and each fluent is completely defined by a fixed subset of these actions.

In contrast to ALTL, in which no two action propositions can be true at the same instant, it is clear that two fluents can hold simultaneously. In FLTL, we can concisely express the *NOARTIFACTS* property of section 3.1, when the screen is initially blanked, as:

$TUNING \equiv \langle \{tune\}, \{endtune\} \rangle \quad Initially_{TUNING} = \text{false}$
$BLANKED \equiv \langle \{blank\}, \{unblank\} \rangle \; Initially_{BLANKED} = \text{true}$

$NOARTIFACTS \equiv \mathbf{G}(TUNING \Rightarrow BLANKED)$

4.1 Actions and Fluents

Fluents have been carefully designed to provide a uniform framework supporting both action- and fluent-based property specifications, as well as their combination. Specifically, every action a implicitly defines a fluent whose initial set of actions is the singleton $\{a\}$ and whose terminating set contains all other actions in the alphabet of the system $A \subseteq Act$:

$Fluent(a) \equiv \langle \{a\}, A - \{a\} \rangle \quad Initially_a = \text{false}$

From the definition, it should be clear that $Fluent(a)$ becomes true the instant a occurs and becomes false with the first occurrence of a different action. We can thus define properties that combine actions with fluents.

It is often more succinct in defining properties to declare a fluent implicitly for a set of events as in:

$Fluent(S) \equiv \langle S, A - S \rangle \quad Initially_S = \text{false}$
where $S = \{a_0, a_1, \ldots a_n\}$

This is equivalent to $a_0 \vee a_1 \vee \ldots \vee a_n$ where a_i represents the implicitly defined $Fluent(a_i)$. Set-based fluents also lead to fewer fluent automata generated during model checking (section 5).

In the next sections, we outline how FLTL model checking is implemented in the LTSA tool, and give examples of using the LTSA concrete syntax for fluents and properties.

5. MODEL CHECKING FLTL

Let M be a finite-state system whose executions define infinite words over $2^{\wp}$, where $\wp$ is a set of propositions. As mentioned, the standard procedure for model checking M against some LTL property φ over $\wp$, consists of the following steps:

1. construct a Büchi automaton B for $\neg\varphi$;
2. check emptiness of the *synchronous* product of B with M (corresponds to the intersection of the two automata.

In our framework, we need to model check LTSs against specifications expressed in FLTL. However, the words produced by executions of LTSs are defined over the set of actions Act, whereas the properties are defined over the set of fluent propositions Φ. This apparent mismatch is not a real problem since, as observed in section 4, an infinite word over actions also defines an FLTL interpretation. We are therefore able to define a procedure for checking FLTL properties of LTSs, without extending the LTS model.

To bridge the gap between state-based models and event-based models such as LTSs, De Nicola and Vaandrager introduced doubly labeled transition systems (DTSs) [9], which were used by Ramakrishna and Smolka in [30]. DTSs have actions labeling transitions and propositions labeling states and thus provide a concise formalism for relating actions to state. In our approach, fluents define state labels implicitly. Naturally, fluents cannot be used to express arbitrary state predicates. However, since they are defined separately, the model of the system remains a simple LTS. In addition, we have developed an FLTL model checking procedure that is based only on LTSs. As such, it requires little modification to the existing implementation of the LTSA tool, which has efficient representations and algorithms for manipulating LTSs. Our model checking procedure avoids the need for augmenting LTSA with a synchronous product operation. It consists of constructions that are generic, and that could therefore be introduced in other LTS based analysis tools.

5.1 Generating a Büchi automaton

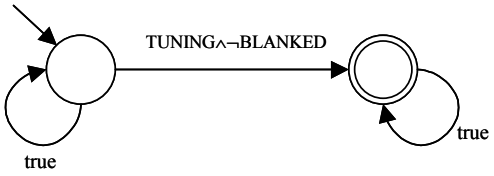

Figure 2. Büchi automaton for ¬G(TUNING ⇒ BLANKED)

Let φ be an FLTL property for a system. The first step of our approach is to generate a Büchi automaton for $\neg\varphi$. We use the algorithm LTL2BUCHI[13], which is an improvement over previous tableau-based constructions by Gerth, Peled, Vardi and Wolper [11] and Daniele, Giunchiglia and Vardi[8]. The result is a Büchi automaton whose alphabet Σ is the powerset of fluents 2^{Φ}. For example, Figure 2 depicts the Büchi automaton obtained for the formula $\neg\mathbf{G}(TUNING \Rightarrow BLANKED)$. In our illustrations of automata, a label $x \in 2^{\Phi}$ is represented as a conjunction of terms, with one term p_i for each fluent $Fl_i \in \Phi$, such that $p_i = Fl_i$ if $Fl_i \in \Phi$, and $p_i = \neg Fl_i$ otherwise. Also, a transition labeled "true" can be fired irrespective of the current values of fluents. For the automaton of Figure 2, each "true" transition is an abbreviation for a transition for each label in 2^{Φ}: $(\neg TUNING \wedge \neg BLANKED)$, $(\neg TUNING \wedge BLANKED)$, $(TUNING \wedge \neg BLANKED)$, and $(TUNING \wedge BLANKED)$.

5.2 Adding Fluent Labels

Let $M = (M_1 \| \cdots \| M_m)$ be a system with alphabet A_M, which consists of a set of m processes $M_1 \ldots M_m$. In order to model check M, we augment it by adding to each composite state a set of self-transitions labeled with the values from 2^{Φ} of fluents that hold at

that state. This is achieved by the parallel composition of *M* with a set of fluent automata.

Fluent automata

For each fluent $Fl \equiv \langle I_{Fl}, T_{Fl} \rangle$ *Initially*$_{Fl}$ where $I_{Fl}, T_{Fl} \subset Act$, we construct a *fluent automaton*, which is an LTS $F_{Fl} = \langle Q, A, \delta, q_0 \rangle$ defined as follows:

- $Q = \{q_t, q_f\}$
- $A = I_{Fl} \cup T_{Fl} \cup 2^{\Phi}$
- $\delta = \{\forall a \in I_{Fl} \mid (q_f, a, q_t), (q_t, a, q_t)\}$
  $\cup \{\forall a \in T_{Fl} \mid (q_f, a, q_f), (q_t, a, q_f)\}$
  $\cup \{\forall x \in 2^{\Phi}, Fl \notin x \mid (q_f, x, q_f)\}$
  $\cup \{\forall x \in 2^{\Phi}, Fl \in x \mid (q_t, x, q_t)\}$
- $q_0 = q_f$ if *Initially*$_{Fl}$ = false, else $q_0 = q_t$

The fluent automaton for the fluent *TUNING* from section 3 is depicted in Figure 3, where ¬*TUNING* is an abbreviation for the set of labels {¬*TUNING*∧¬*BLANKED*, ¬*TUNING*∧*BLANKED*} and *TUNING* is an abbreviation for the set of labels {*TUNING*∧¬*BLANKED*, *TUNING*∧*BLANKED*}.

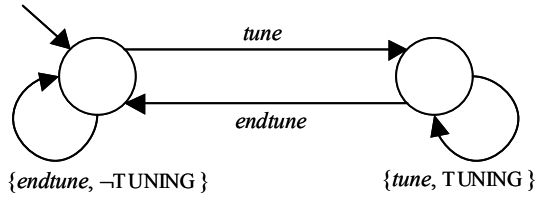

Figure 3. Fluent automaton for TUNING initially false.

The automaton has two states, one in which the fluent does not hold and one in which it does. The initial state of the automaton is determined by the value of the attribute *Initially*$_{Fl}$. The automaton moves into the holding state when an action from its initiating set occurs and to the not holding state when an action from its terminating set occurs. In other words, the automaton relates actions to the truth-value of the fluent and those values are represented by the self-transitions labeled from 2^{Φ}. So if $Fl_1 \cdots Fl_n$ represent the fluents in Φ, the composition

$$MF = (M_1 \| \cdots \| M_m \| F_{Fl_1} \| \cdots \| F_{Fl_n})$$

achieves the desired effect of adding to each composite state in *M*, self-transitions labeled with the fluents that hold in that state. Effectively these transitions simulate the state labeling of DLTs. Note that for model checking, the composition of the system and the fluent automata can be explored on-the-fly.

5.3 Synchronous product

As discussed, model checking involves computing the synchronous product of the system with a Büchi automaton. In our context, this product requires that for each step that system *M* takes (i.e. transition labeled with an action from A_M), the Büchi automaton B_Φ over 2^{Φ} also take a step according to the fluent values resulting from that transition. These fluent values, as described above, are represented by self-transitions on each state of the augmented system *MF*. To compute the synchronous product, we use conventional parallel composition and a synchronizer automaton constructed as follows.

Synchronizer automaton

For a set of fluents Φ and a system with alphabet A_M, such that $(\forall Fl \in \Phi, I_{Fl} \subseteq A_M \wedge T_{Fl} \subseteq A_M)$, the synchronizer automaton is an LTS $Sync_{A_M \Phi} = \langle Q, A, \delta, q_0 \rangle$ where:

- $Q = \{q_0, q_1\}$
- $A = A_M \cup 2^{\Phi}$
- $\delta = \{\forall a \in A_M \mid (q_0, a, q_1)\} \cup \{\forall x \in 2^{\Phi} \mid (q_1, x, q_0)\}$

Figure 4 depicts the synchronizer for the example formula ¬**G**(*TUNING* ⇒ *BLANKED*) for a system alphabet {*tune*, *endtune*, *blank*, *unblank*, *anon*} where *anon* is some other action that the system performs but is not used in defining fluents.

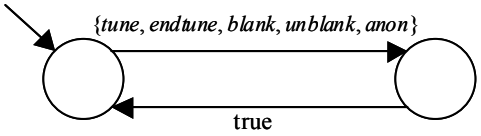

Figure 4. Synchronizer automaton

The required synchronous product between the augmented system *MF* with Büchi automaton B_Φ can now be formed by the following parallel composition:

$$MFB = (M_1 \| \cdots \| M_m \| F_{Fl_1} \| \cdots \| F_{Fl_n} \| Sync_{A_M \Phi} \| B_\Phi)$$

Following [12], we mark accepting states in B_Φ by adding a self-transition labeled *@B*. In essence, after every transition of system *M*, the synchronizer automaton forces the Büchi automaton B_Φ to observe the fluent values in the resulting state and take a step accordingly.

5.4 Model checking

For model checking, since B_Φ accepts the complement of the language accepted by the FLTL property formula, we need to check that the language accepted by *MFB* is empty. This consists of checking that *MFB* does not contain any strongly connected component that is reachable from the initial state and contains an accepting state. We use Tarjan's depth first search algorithm [19] for computing strongly connected components. *MFB* is not explicitly constructed but explored "on-the-fly".

The LTSA tool includes a technique that combines an assumption of fair choice of transitions with the use of action priority to specify scheduling conditions[14]. This technique requires computing terminal sets of states, i.e., strongly connected components in which there are no transitions from a state in the component to a state outside the component. To be compatible with this technique, the Büchi automata that we use for model-checking must be complete [12]. A Büchi automaton B_Φ over 2^{Φ} is complete if in every state there is an enabled transition for every label in 2^{Φ}. An automaton B_Φ can be easily completed by adding a non-accepting "sink" state s_k, and then introducing transitions to

s_k for all undefined transitions in each state of the automaton. The Büchi automaton of Figure 2 is complete, so it does not require this additional "sink" state.

6. TESTER AUTOMATON

The procedure outlined in the previous section has a number of deficiencies: the use of a synchronizer automaton and parallel composition to form the synchronous product doubles the number of states that must be explored during model checking, and representing fluent values by sets of transitions causes an unnecessarily large number of transitions to be explored. Moreover, the problem of dealing with unobservable actions as mentioned in section 3 is not addressed. To deal with these issues, we partition the composition to form a "tester automaton". This is a Büchi automaton B_{AM} that recognizes infinite words over the alphabet $A_M \subseteq Act$ of the system.

The composition *MFB* which we constructed in the previous section can be factored as follows:

$$B_{AM} = (F_{Fl_1} \| \cdots \| F_{Fl_n} \| Sync_{A_M \Phi} \| B_\Phi) \setminus 2^\Phi$$

$$M = (M_1 \| \cdots \| M_m)$$

We form the "tester" automaton B_{AM} from the parallel composition of the fluent automata, the synchronizer and the Büchi automaton B_Φ. Since the alphabet A_M of the system M does not contain any of the labels in 2^Φ, we can safely hide these actions in the composition B_{AM}. It is now possible to explicitly construct the LTS for B_{AM} and then optimize it before composing it with the system.

Firstly, we remove the unnecessary intermediate states in which the Büchi automaton has not yet observed the state resulting from a transition on some action in A_M. This is performed by coalescing states separated by the τ transitions caused by hiding the labels in 2^Φ. Specifically, due to the use of a synchronizer, any execution of B_{AM} will consist of a sequence:

$$q_0 \xrightarrow{a_0} q_1 \xrightarrow{\tau} q_2 \xrightarrow{a_1} q_3 \xrightarrow{\tau} \cdots, \text{where } a_0, a_1, \ldots \in A_M$$

To eliminate intermediate states of B_{AM}, for any two consecutive transitions (q_i, a, q_j),(q_j, τ, q_k) in its LTS, we first replace these transitions with a single transition (q_i, a, q_k), and then remove the resulting unreachable state q_j.

Subsequently, we minimize the resulting LTS with respect to strong bisimulation. The tester automaton obtained is a Büchi automaton that recognizes sequences of actions from the alphabet A_M. As before, accepting states in B_{AM} are marked by a self-transition labeled *@B*.

Although there is a danger of intermediate state-space explosion in computing the tester automaton B_{AM}, it has the following advantages. Significant reductions are obtained if the initiating and terminating sets of different fluents are not disjoint. For example, this is always the case for the terminating sets of action fluents. In addition, the cost of a "step" in exploring the state space of *MFB* is proportional to the number of automata in the composition since in each composite state we need to look at which actions are enabled in each automaton. Consequently, computing and optimizing B_{AM} speeds up the model checking.

Figure 5 illustrates the tester automaton generated by the LTSA for $\neg$**G**(*TUNING* $\Rightarrow$ *BLANKED*) with alphabet {*tune*, *endtune*, *blank*, *unblank*, *anon*}.

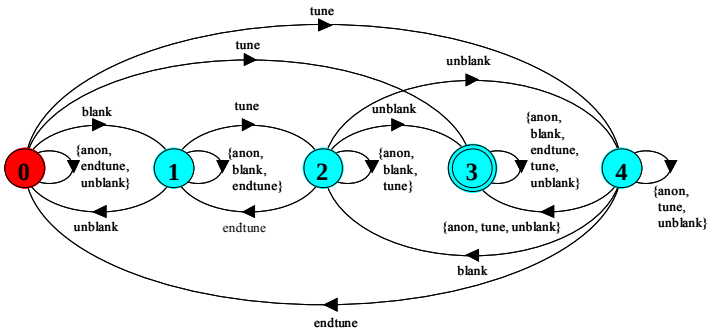

Figure 5. Tester automaton for ¬G(TUNING ⇒ BLANKED)

6.1 Safety Properties

Following the formal definition from Alpern and Schneider [2] and the approach presented in [3], a safety property stipulates that some "bad thing" does not happen. If a "bad thing" happens in an infinite sequence, then it must also do so after some finite prefix and must be irremediable. In the case of a tester process, which recognizes violating sequences as it is generated from the negation of a property, a finite sequence of actions is recognized if it ends in an accepting state and there is no action that can cause the tester process to leave that accepting state. We call such accepting states *terminal*. State 3 of Figure 5 is an example of a terminal accepting state.

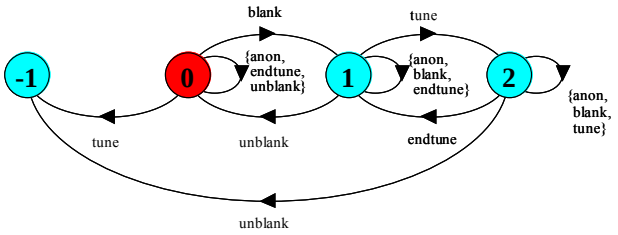

Figure 6. Deterministic tester for ¬G(TUNING ⇒ BLANKED)

If all the accepting states of a tester automaton are terminal then the property it represents is a pure safety property. We can replace the accepting states with ERROR states, in which case model checking reduces to a simple reachability search for ERROR states[5]. Further, since the tester automaton is only required to detect finite sequences, we can usually further minimize the tester automaton using trace equivalence, which involves standard determinization and minimization for finite state automata[17]. Applying this to the example of Figure 5 results in the tester automaton of Figure 6, in which state (-1) is the ERROR state.

6.2 Partial Order Reduction

The tester automaton as described so far is constructed with respect to the alphabet $A_M \subseteq Act$ of the system, since A_M is used in defining the synchronizer automaton and fluent automata. This has two disadvantages: firstly, we must reconstruct the tester when we wish to check a system with a different alphabet even when the

actions we use to define fluents do not change and secondly, more seriously, it causes a problem with partial order reduction. This latter problem is clearly articulated by Valmari[33]. In essence, synchronizing the tester automaton with every action of the system makes all transitions dependent on each other and, consequently, prevents any reduction obtainable from the use of independency in computing ample sets[28] during partial order reduction. In our example tester depicted in Figures 5 & 6, in which the action *anon* represents the set of system actions not directly used in the definition of fluents, it can be seen that a self-transition labeled with this action is enabled in every state.

To address these problems and also deal with τ transitions in the system, we represent all actions in the system (including action τ) that are not used in the definition of fluents, by a single action "*". If A_F is the set of actions appearing in fluent definitions, then the alphabet used in the construction of the synchronizer is $A_F \cup \{*\}$. Steps on action * in the synchronizer represent stuttering steps in the system with respect to fluent values, but since such stuttering may be infinite, they must be taken into account in model checking. The resulting synchronizer is then used in the construction of the tester automaton, which is performed as defined earlier in this section. However, to allow the tester to observe actions associated with "*" we define a slightly modified parallel composition operator "$\|_*$", as follows.

For a system $M = \langle Q_M, A_M, \delta_M, q_{0M} \rangle$ and a tester automaton $T = \langle Q_T, A_T, \delta_T, q_{0T} \rangle$ such that $A_T - \{*\} \subseteq A_M$, the modified parallel composition $M\|_*T$ is an LTS $\langle Q, A, \delta, q_0 \rangle$, where $Q = Q_M \times Q_T$, $A = A_M$, $q_0 = (q_{0M}, q_{0T})$, and δ is defined as follows, where a is an observable action or τ:

- $$\frac{M \xrightarrow{a} M', T \xrightarrow{a} T'}{M \|_* T \xrightarrow{a} M' \|_* T'}$$ (note that T has no τ transitions)
- $$\frac{M \xrightarrow{a} M', T \xrightarrow{*} T', \ a \notin A_T}{M \|_* T \xrightarrow{a} M' \|_* T'}$$

Similarly to the conventional parallel composition operator, operator "$\|_*$" synchronizes transitions labeled with common actions in the alphabets of the two LTSs. Additionally, transitions in system M labeled with actions in $(A_M - A_T)$ or τ synchronize with * transitions in T, if such transitions are enabled. Since T is a complete automaton, the second rule will always apply when the first one does not, so the tester observes every step of the system. The modified operator is not meant to be commutative or associative since it is only used to combine tester with system.

An advantage of tester automata that use * transitions is that they are independent of the system, and can be reused across systems with the same set of fluents. Moreover, it is now possible to introduce an optimization to alleviate the problem discussed above related to partial order reduction. Let q be a state in T that has a self-transition $(q, *, q)$, such that there exists no transition from q to a different state labeled with *. The same effect of any such transitions can be achieved by removing the transition from the tester T, and adding the following rule to the definition of operator "$\|_*$":

- $$\frac{M \xrightarrow{a} M', T \neg\xrightarrow{*}, a \notin A_T}{M \|_* T \xrightarrow{a} M' \|_* T}$$

where the notation $T \neg\xrightarrow{*}$ means that there is no transition labeled with * enabled in T. The result of removing such transitions weakens the dependence between M and T, and permits partial order reduction. Partial order reduction is only enabled in LTSA for LTL-X properties. These are properties that do not contain operator **X** and are therefore closed under stuttering [19], as required for partial order reduction.

7. FLTL in the LTSA

The Labelled Transition System Analyser (LTSA) is a general purpose tool for exploring event-based system specifications. Systems are described using FSP, which is a process algebra notation for describing finite state processes. In addition, models can be constructed from a set of scenarios described by message sequence charts. The tool supports interactive model exploration, domain specific animation[23], compositional reachability analysis and verification with respect to safety properties specified as automata[5] and progress properties specified as action sets[14]. The tool has now been augmented with the FLTL model-checking procedure described in the previous section. In the following, we use examples to illustrate the use of the FLTL features.

7.1 Fluents and FLTL properties

The concrete syntax for FLTL formulas used in the LTSA follows as closely as possible the LTL syntax used in SPIN. The following operators are defined:

Unary operators (*unop*):		Binary operators(*binop*):	
`[]`	always (**G**)	`U`	strong until
`<>`	eventually (**F**)	`W`	weak until
`X`	next time	`&&`	logical AND
`!`	logical negation	`||`	logical OR
		`->`	implication
		`<->`	equivalence

Note that we will be representing logical OR as $\|_{OR}$ here to avoid confusion with the parallel composition operator. An FLTL formula is defined as Φ := `True` | `False` | *prop* | (Φ) | *unop* Φ | Φ *binop* Φ, where *prop* is a fluent, an action or a set of actions as defined in section 3. Fluents are specified as shown by a pair of actions (or actions sets). An initialization clause is optional. If missing, the fluent is assumed to be initially false.

The concrete syntax for the property NOARTIFACTS introduced in section 3.1, for the case where the screen is initially blanked is:

```
fluent BLANKED =
       <blank,unblank> initially True

fluent TUNING  = <tune,endtune>
assert NOARTIFACTS = [](TUNING -> BLANKED)
```

Note that, since the screen is initially blanked, we need to explicitly set the value of fluent *BLANKED* to true. In addition, we can define an equivalent property with actions:

```
assert ActionNA =
     [](((unblank -> (!tune W blank))
        && (tune -> (!unblank W endtune)))
```

A simple liveness property that asserts that the screen is always eventually unblanked can be expressed using the action *unblank*:

```
assert UNBLANK = []<>unblank
```

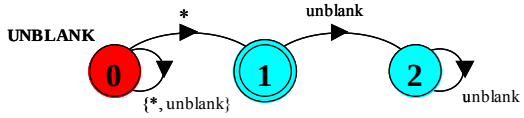

Figure 7. Tester automaton for `[]<>unblank`

The tester automaton generated from this property is shown in Figure 7. Note that the tester corresponds to the negation of the property, as required by model checking. The automaton illustrates that * actions are necessary in state 0. They allow the tester to non-deterministically move to the accepting state, that corresponds to the moment that *unblank* stops occurring for ever. An * self-transition at state 1 has been removed by our optimization. With our modified parallel composition operator, if at that stage the system executes an infinite sequence of actions that are not *unblank*, this will be detected as a property violation in the composition of the system with the tester.

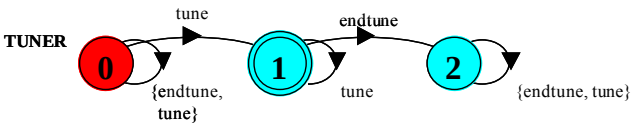

Figure 8. Tester automaton for `[](tune -> <>endtune)`

In contrast, the tester generated for the response property:

```
assert TUNER = [](tune -> <>endtune)
```

depicted in Figure 8 has no * transitions. In this case, only the action *tune* can take the automaton into the accepting state. Again, the automaton stays in the accepting state if the system executes an infinite sequence of actions that are not *endtune*.

Finally, it is possible to combine properties as in:

```
assert SCREEN  = (NOARTIFACTS && UNBLANK)
```

7.2 Indexed Fluents

We have found it convenient to declare indexed sets of fluents. For example, *CRITICAL[i]* is true when process *p[i]* enters a critical section and false when it exits:

```
const  N = 4
fluent CRITICAL[i:1..N] =
       <p[i].enter, p[i].exit>
```

Mutual exclusion between two processes p[1] and p[2] can simply be expressed as:

```
assert MUTEX =
       []!(CRITICAL[1] && CRITICAL[2])
```

We can also express a mutual exclusion property for N processes:

```
assert MUTEX_N =
  []!(allOr[i:1..N-1]
     (CRITICAL[i] && CRITICAL[i+1..N]))
```

Here we use an *or replicator* where:

$$\text{allOr}[i{:}1..N]\ C[\text{i}] \equiv C[1] \|_{OR} \ldots \|_{OR} C[N]$$

Moreover, the LTSA supports fluent propositions of the form:

$$FL[i{:}1..N] \equiv FL[1] \|_{OR} \ldots \|_{OR} FL[N]$$

An *and replicator* is used to define the required liveness property:

```
assert LIVE_MUTEX =
       allAnd[i:1..N] []<>CRITICAL[i]
```

The use of index ranges allows the concise expression of properties. For example, the safety property for a system of readers and writers in which either multiple readers or a single writer can access a shared resource can be expressed as:

```
const N = 3

fluent READING[i:1..N] =
       <startRead[i], endRead[i]>
fluent WRITING[i:1..N] =
       <startWrite[i], endWrite[i]

// readers and writers cannot access at the same time
assert READ_WRITE =
  []!(READING[1..N] && WRITING[1..N])

// only a single writer can access at one time
assert ONE_WRITE =
    []!allOr[i:1..N-1]
      (WRITING[i] && WRITING[i+1..N])

// the required Readers-Writers safety property is
assert RW_SAFE = (READ_WRITE && ONE_WRITE)
```

Table 1 - Tester automata size (#states)

Property	*N*	B_Φ	B_A	*Safe*(B_A)
MUTEX_N	2	2	5	4
	3	2	9	5
	4	2	17	6
	5	2	33	7
LIVE_MUTEX	2	6	8	-
	3	8	11	-
	4	10	14	-
	5	12	17	-
RW_SAFE	2	4	22	7
	3	4	74	12
	4	4	769	21
	5	4	1058	38

Table 1 displays the size of the tester automata for the above properties for different values of *N*. The column headed B_Φ gives the number of states of the Büchi automata generated by LTL2BUCHI, the one labeled B_A gives the number of states of the corresponding tester automata and *Safe*(B_A) the number of states after making safety properties deterministic and minimal. In the case of safety properties, the table indicates that computing tester automata may suffer from intermediate state-space explosion, as mentioned in section 6. However, this is compensated by the

reduction achieved by minimization according to trace equivalence. Note that for a value of *N*=5, the *RW_SAFE* property requires 10 fluent automata for its construction (i.e. *READING*[1..5] and *WRITING*[1..5]). Consequently, where possible, it is worth checking safety properties independently from liveness properties since the safety property reduction cannot be applied to combined properties, for example to (*LIVE_MUTEX && MUTEX_N*).

7.3 Counterexamples

Although tester automata retain no information concerning the fluents from which they are formed, we have found it provides useful diagnostic information to reconstruct the value of fluents from error traces when displaying counterexamples. This is accomplished by computing fluent values after each action in the trace. The example below is a counter example produced from the MUTEX property. The trace is annotated with the names of fluents that are true at the time an action occurs:

```
Trace to property violation in MUTEX:
    p[1].mutex.down
    p[1].enter      CRITICAL[1]
    p[2].mutex.down  CRITICAL[1]
    p[2].enter   CRITICAL[1] && CRITICAL[2]
Analysed in: 20ms
```

8. CONCLUSION

The usefulness of LTL to specify properties in event-based system models is acknowledged by Leuschel, Massart and Currie[24] in the context of CSP. They take the approach outlined in section 3 of associating primitive propositions with events. Paun and Chechik[27] make a convincing case for supporting events in LTL specifications of state-based system models. In this paper, we describe the use of *fluents* as a means of including propositions related to state in LTL properties of event-based systems. Fluents permit the concise specification of properties concerned with state. They provide an elegant way of specifying properties that combine propositions over the occurrence of actions or events with propositions over the states that these actions bring about. Moreover, the paper has outlined an approach for model checking FLTL properties, based on the use of Büchi automata.

The FLTL model-checking approach presented should be applicable to other LTS based analysis tools. The inefficient representation of fluent values as sets of transitions is compensated for by the construction of tester automata that remove these transitions before model checking. The formation of a tester automaton also permits the application of optimizations such as the recognition and minimization of safety properties. This procedure relies on the LTL to Büchi automata translation procedure producing automata with only terminal accepting states for safety properties. This will not be the case for all safety properties, however, our experience so far is that it applies to a sufficiently large class of properties, in particular invariants, to make it worthwhile.

Critical to the success of any LTL model-checking procedure is the use of partial-order reduction for those properties that are closed under stuttering. We have outlined how the use of the ||* operator permits partial order reduction by weakening the dependence between tester actions and other system actions. The use of this operator has the additional advantage that it makes tester construction independent of the systems to which they are applied. In other words, we can modify the system model without rebuilding the tester so long as the actions associated with the property remain in the alphabet of the system. Space has allowed us to only sketch the relationship with partial order reduction. At the moment, when exploring the state space during model checking, we take the conservative approach of fully expanding any state in which an "*" action is enabled. We are currently exploring conditions under which this can be relaxed to achieve more reduction.

In conclusion, we have used fluents as a way of reasoning about state in event based system models. We are already finding other uses for fluents; in specifying preconditions for scenarios and in specifying performance measures for stochastic models.

Acknowledgements

We are grateful for helpful discussions on the ideas contained within this paper with Jeff Kramer and Sebastian Uchitel. The UK author is gratefully for support from EPSRC grants READS GR/S03270/01 and AEDUS GR/R95715/01.

9. REFERENCES

[1] R. Allen and D. Garlan, *A Formal Basis for Architectural Connection*, ACM Transactions on Software Engineering and Methodology (ACM TOSEM), Vol. 6, No. 3, pp. 213-249, 97.

[2] B. Alpern and F. Schneider, *Defining Liveness*, Information Processing Letters, Vol. 21, No. Oct, pp. 181-185, 1985.

[3] B. Alpern and F. B. Schneider, *Recognising Safety and Liveness,* Department of Computer Science, Cornell University, TR 86-727, January 1986.

[4] M. Bernardo, P. Ciancarini and L. Donatiello, *Architecting Software Systems with Process Algebras,* University of Bologna, UBLCS-2001-7, July 2001.

[5] S. C. Cheung and J. Kramer, *Checking Subsystem Safety Properties in Compositional Reachability Analysis*, 18th International Conference on Software Engineering (ICSE'18), Berlin, Germany, pp. 144-154, March 1996.

[6] E. M. Clarke and E. A. Emerson, *Synthesis of synchronisation skeletons for branching time temporal logic*, Logic of Programs Workshop, Yorktown Heights NY, 131.

[7] R. Cleaveland, J. Parrow and B. Steffen, *The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems*, ACM Transactions on Programming Languages and Systems, Vol. 15, No. 1, pp. 36-72, 93b.

[8] M. Daniele, F. Giunchiglia and M. Y. Vardi, *Improved Automata Generation for Linear Temporal Logic*, 11th International Conference on Computer Aided Verification (CAV 1999), Trento, Italy, 1633, July 1999.

[9] R. De Nicola and F. W. Vaandrager, *Three Logics for branching bisimulation*, Journal of the ACM, Vol. 42, No. 2, pp. 458-487, 1995.

[10] M. Dwyer, G. Avrunin and J. Corbett, *Patterns in property Specifications for Finite-State Verification*, 21st International Conference on Software Engineering (ICSE'99), Los Angeles, CA, pp. 411-420, 16-22 May 1999.

[11] R. Gerth, D. Peled, M. Y. Vardi and P. Wolper, *Simple On-the-fly Automatic Verification of Linear Temporal Logic*, 15th IFIP/WG6.1 Symposium on Protocol Specification, Testing and Verification (PSTV'95), Warsaw, Poland, pp. 3-18, June 1995.

[12] D. Giannakopoulou, *Model Checking for Concurrent Software Architectures,* PhD Thesis, Imperial College London, 1999.

[13] D. Giannakopoulou and F. Lerda, *From States to Transitions: Improving translation of LTL formulae to Büchi automata*, 22nd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2002), Houston, Texas, November 2002.

[14] D. Giannakopoulou, J. Magee and J. Kramer, *Checking Progress with Action Priority: Is it Fair?*, 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'99), Toulouse, France, 1687, pp. 511-527, September 1999.

[15] C. A. R. Hoare, *Communicating sequential processes*, Englewood Cliffs, N.J. ; London, Prentice-Hall International, 1985.

[16] G. J. Holzmann, *The Model Checker SPIN*, IEEE Transactions on Software Engineering, Vol. 23, No. 5, pp. 279-295, 97.

[17] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 79.

[18] R. A. Kowalski and M. Sergot, *A Logic-Based Calculus of Events*, New Generation Computing, Vol. 4, No., pp. 67-95, 1986.

[19] L. Lamport, *The Temporal Logic of Actions*, ACM Transactions on Programming Languages and Systems, Vol. 16, No. 3, pp. 872-923, 94.

[20] J. Magee, N. Dulay, S. Eisenbach and J. Kramer, *Specifying Distributed Software Architectures*, 5th European Software Engineering Conference (ESEC'95), Sitges, Spain, 989, pp. 137-153, September 1995.

[21] J. Magee and J. Kramer, *Concurrency - State Models & Java Programs*, Chichester, John Wiley & Sons, 1999.

[22] J. Magee, J. Kramer and D. Giannakopoulou, *Behaviour Analysis of Software Architectures*, 1st Working IFIP Conference on Software Architecture (WICSA1), San Antonio, TX, USA, 22-24 February 1999.

[23] J. Magee, N. Pryce, D. Giannakopoulou and J. Kramer, *Graphical Animation of Behavior Models*, ICSE, Limerick, June 2000.

[24] Michael Leuschel, Thierry Massart and A. Currie, *How to Make FDR Spin: LTL Model Checking using Refinement*, Proceedings of FME'2001, LNCS 2021, pp. 99-118.

[25] R. Miller and M. Shanahan, *The Event Calculus in Classical Logic - Alternative Axiomatisations*, Linkoping Electronic Articles in Computer and Information Science, Vol. 4, No. 16, pp. 1-27, 1999.

[26] R. Milner, *Communication and Concurrency*, Prentice-Hall, 89.

[27] D. O. Paun and M. Chechik, *Events in Linear-Time Properties*, Proceedings of 4th International Conference on Requirements Engineering, Toronto, June 1999.

[28] D. Peled, *Combining Partial Order Reductions with On-the-Fly Model Checking*, 6th International Conference on Computer Aided Verification (CAV'94), Stanford, California, 818, pp. 377-390, June 1994.

[29] J.-P. Queille and J. Sifakis, *Specification and verification of concurrent systems in CESAR*, 5th International Symposium on Programming, Turin, 137, pp. 337-350, April 6-8 1982.

[30] Y. S. Ramakrishna and S. A. Smolka, *Partial-Order Reduction in the Weak Modal Mu-Calculus*, Proceedings of the Eighth International Conference on Concurrency Theory (CONCUR '97), Warsaw, Poland, LNCS 1243, pp. 5-24, July 1997.

[31] A. W. Roscoe, *A Classical Mind: Essays in Honour of C.A.R. Hoare*, pp. 353-378, Prentice-Hall, 94.

[32] E. Sandewall, *Features and Fluents: The Representation of Knowledge about Dynamical Systems*, Oxford University Press, 1994.

[33] A. Valmari, *On-the-fly Verification with Stubborn Sets*, Proceedings of CAV '93, 5th International Conference on Computer-Aided Verification, Elounda, Greece, LNCS 697, pp. 397-408.

[34] R. van Ommering, *Horizontal Communication: a Style to Compose Control Software,* Philips Research Laboratories.

[35] M. Y. Vardi and P. Wolper, *An automata-theoretic approach to automatic program verification*, 1st Symposium on Logic in Computer Science, Cambridge, pp. 322-331, June 1986.